On the Relationship Between Algebraic Module Specifications and Program Modules *

Michael Löwe, Hartmut Ehrig, Werner Fey
Technische Universität Berlin, Franklinstraße 28/29, 1000 Berlin 10
Dean Jacobs
University of Southern California, Los Angeles, CA 90089-0782

Abstract

This paper studies the relationship between our long-standing algebraic concept of module specifications and modules as they appear in conventional programming languages. The approach we take is to introduce an intermediate algebraic concept of *abstract program modules*, which structurally model concrete program modules. We show how a system of abstract program modules is formally related to a system of module specifications. This work is intended to aid the design of modularization mechanisms and to facilitate the transformational development of a system of module specifications into a system of program modules.

1 Introduction

Over the last ten years, we have developed an algebraic concept of module specifications [BEPP87, EM90] that serves as a basis for the software development process. This concept was mainly influenced by contributions of Parnas [Par72b, Par72a] to datatypes and modules. Structurally, these module specifications are somewhat different from the modules appearing in programming languages such as Modula2 [Wir82], Ada [Spr81], and ML [HMM86] [1]. This raises a number of interesting questions concerning the relative expressive power of these concepts.

- Can every system of module specifications be implemented as an equivalently structured system of program modules?
- Can every system of program modules be described as an equivalently structured system of module specifications?

This paper studies the relationship between module specifications and program modules with these questions in mind. The approach we take is to introduce an intermediate

*This work was carried out as part of a long term cooperation agreement between the Technical University of Berlin and the University of Southern California.

[1] In contrast, the modularization mechanisms of the specification languages OBJ [FGJM85], PLUSS [Gau84], and Extended ML [ST86] more closely resemble the modularization mechanisms of programming languages.

algebraic concept of *abstract program modules*, which structurally model concrete program modules. We show how a system of abstract program modules can be directly translated into a system of module specifications, answering the second question above in the affirmative.

There are three significant structural differences between module specifications and program modules. First, the import part of a module specification simply describes its required resources, as in most "module interconnection languages" [PDN82], rather than naming specific modules to provide those resources. This enhances reusability in that the same high-level module can be used with different low-level modules in different contexts. Second, compound module specifications are built up using a variety of operations, such as composition and union. Such operations clarify the structure of a system and facilitate its restructuring using algebraic laws. Third, program modules generally provide separate import parts for the export part and the body. In contrast, a module specification has a single import part that services its body and, indirectly, also its export part. It is notable that this is not a significant limitation in the context of this paper: every system of abstract program modules can be directly translated into a system of module specifications. The results reported in this paper are still somewhat preliminary, however, in that our current formulation of abstract program modules does not support the notion of generic parameterization.

This paper is organized as follows. Section 2 introduces abstract program modules by way of an informal comparison with concrete program modules. Section 3 presents some mathematical preliminaries. Section 4 defines abstract program modules. Section 5 introduces module specifications and relates them to abstract program modules. Section 6 presents some concluding remarks concerning the extension of abstract program modules to include generic parameterization.

2 From Concrete to Abstract Program Modules

This section introduces abstract program modules by way of an informal comparison with modules in the programming language Modula2. There are two kinds of modules in Modula2, *definition* modules, which introduce collections of resources, and *implementation* modules, which implement resources introduced in definition modules. As an example, the following definition and implementation modules provide a bounded buffer type `Buffer` and its associated operations.

```
DEFINITION MODULE BufferMod;
   FROM ElementMod IMPORT Element;
   TYPE Buffer;
   PROCEDURE initialize(VAR s:Buffer);
   PROCEDURE insert(VAR s:Buffer; c:Element);
   PROCEDURE remove(VAR s:Buffer; VAR c:Element);
   PROCEDURE isfull(s:Buffer):BOOLEAN;
   PROCEDURE isempty(s:Buffer):BOOLEAN;
END BufferMod.

IMPLEMENTATION MODULE BufferMod;
   FROM ElementMod IMPORT Element;
```

```
   CONST max = 100;
   TYPE Buffer = RECORD
                 data:ARRAY[0..max-1] OF Element;
                 front:CARDINAL;
                 size:CARDINAL;
                 END;

   PROCEDURE initialize(VAR s:Buffer);
      BEGIN
      s.front := 0;
      s.size := 0
      END initialize;

   PROCEDURE insert(VAR s:Buffer; c:Element);
      BEGIN
      IF NOT isfull(s) THEN
         s.data[(s.first+s.size) MOD max] := c;
         s.size := s.size+1
         END
      END insert;
   ...
END BufferMod.
```

Modules may be used to hide auxiliary operations and constants, such as `max`, and the underlying representation for types, such as `Buffer`. A type is said to be either *opaque* or *transparent* depending upon whether its representation is defined in the implementation module, as in the case of `Buffer`, or in the definition module. Modules are interconnected by import clauses. The import clause in the above example allows `BufferMod` to reference `Element`, but no other identifiers, of `ElementMod`. Alternatively, the clause

```
IMPORT ElementMod
```

would provide "qualified" access to all identifiers exported by `EementMod`, e.g., `Element` would be referred to as `ElementMod.Element`.

Abstract program modules are intended to provide an algebraic model for the *structural* aspects of concrete program modules. As such, they do not model imperative/object-oriented features such as state, variables, and side-effects. Rather, they model the "type view" of programs, i.e., those features that refer to type declarations and implementations, and function declarations and implementations. An *abstract program module* consists of an export part and a body, corresponding to the definition and implementation modules of Modula2. As an example, the following abstract program module models the above bounded buffer modules. Note that we choose a specific syntax here to make the example easier to understand; subsequent sections of this paper use an abstract syntax.

```
MODULE BufferMod
   EXPORT
      IMPORT ElementMod USE Element
             BooleanMod USE Boolean
      SORTS Buffer
```

```
      OPNS
         create  : -> Buffer
         insert  : Buffer, Element -> Buffer
         remove  : Buffer -> Buffer
         read    : Buffer -> Element
         isfull  : Buffer -> Boolean
         isempty : Buffer -> Boolean
   BODY
      IMPORT CardinalMod USE Cardinal
             ArrayMod USE Array(Element), empty, update
             BooleanMod USE true, false
      OPNS
         max : -> Cardinal
         rep : Array(Element), Cardinal, Cardinal -> Buffer
      EQNS
         max = 100
         create = rep(empty,0,0)
         isfull(rep(d,f,s))=false =>
            insert(rep(d,f,s),c) = rep(update(d,(f+s) MOD max,c),f,s+1)
         isfull(rep(d,f,s))=true =>
            insert(rep(d,f,s),c) = rep(d,f,s)
         ...
END
```

Each type in the concrete program module is modeled by a sort in the abstract program module. Sorts corresponding to predefined types such as `CARDINAL` and `ARRAY` are explicitly imported from other modules in the abstract setting. User-defined types such as `Buffer` are implemented using new "constructor operations" such as `rep`. Opaque types are modeled by declaring the associated operations in the body, as in the case of `rep`. Transparent types are modeled by declaring these operations in the export part. Each procedure in the concrete program module is modeled by a collection of operations in the abstract program module, e.g.,

```
PROCEDURE remove(VAR s:Buffer; VAR c:Element);
```

is modeled by

```
OPNS
   remove  : Buffer -> Buffer
   read    : Buffer -> Element
```

Here, imperative updating of variables is modeled as the creation of values in the type view.

Abstract and concrete program modules differ concerning the automatic exportation of imported resources. In Modula2, identifiers imported by a definition module are not automatically exported by that module. This leads to certain semantic complications, in particular, it is possible for a resource to be semantically present even though it is not visible, as the following example illustrates.

```
DEFINITION MODULE N;
```

```
   TYPE T;
END N.

DEFINITION MODULE M;
   FROM N IMPORT T;
   PROCEDURE f(x:CARDINAL):T;
   PROCEDURE g(x:T):CARDINAL;
END M.
```

In a context where `M` is imported but `N` is not, `T` is semantically present, since expressions such as `g(f(1))` are legal, however, `T` is not visible. Moreover, in a context where both `M` and `N` are imported, the range of `M.f` is considered to be the same as `N.T`, thus, an elaborate notion of type equivalence is required.

In contrast, an abstract program module automatically exports all resources that are imported into its export part. References to a resource are always qualified by the name of the module where that resource was *declared.* For example, the sort `Boolean` is automatically exported from `BufferMod` as `BooleanMod.Boolean` and the range of `BufferMod.isfull` is `BooleanMod.Boolean`. Formally speaking, *every* reference to a resource within a module in which that resource is not declared must be qualified by the name of a module in which that resource is declared. Use-lists, e.g., as in `IMPORT ElementMod USE Element`, appear as syntactic sugar in this section to improve the readability of the example.

3 Mathematical Preliminaries

Abstract program modules and module specifications are extensions of the basic notion of algebraic specifications of datatypes [LZ74, GTW76, TWW78, EM85]. An *algebraic specification* $SPEC$ is a triple (S, OP, E) where S is a set of sort declarations, OP is a set of operation declarations, and E is a set of equations. We distinguish between *complete* and *incomplete* algebraic specifications, where (S, OP, E) is complete iff every sort referenced in OP is declared in S and every operation referenced in E is declared in OP. The *union* $SPEC_1 \cup SPEC_2$ (resp. *intersection* $SPEC_1 \cap SPEC_2$) of two algebraic specifications is taken to be the union (resp. intersection) of each of their components. $SPEC_1$ is said to be a *subspecification* of $SPEC_2$, denoted $SPEC_1 \subseteq SPEC_2$, if each component of $SPEC_1$ is a subset of the corresponding component of $SPEC_2$.

Given an algebraic specification $SPEC = (S, OP, E)$, a *$SPEC$-datatype* A consists of a carrier set A_s for each $s \in S$ and an operation N_A over these carrier sets for each $N \in OP$. These operations are required to satisfy all equations in E. The set of all $SPEC$-datatypes forms a domain $Alg(SPEC)$. The *initial* datatype w.r.t. $SPEC$, denoted T_{SPEC}, satisfies the following conditions.

1. The data elements of T_{SPEC} are exactly those that can be generated by application of the operations of $SPEC$.

2. The operations of T_{SPEC} satisfy exactly those properties that follow from the equations of $SPEC$ using equational deduction and structural induction.

We are interested in two mappings involving subspecifications $SPEC_1 \subseteq SPEC_2$. First, for each $SPEC_2$-datatype A there is a $SPEC_1$-datatype $[A]_{SPEC_1}$, called the

$SPEC_1$ *reduct* of A, in which carrier sets and operations of A that are not in $SPEC_1$ are eliminated. Second, there is a *free construction*

$$FREE_{SPEC_1 \to SPEC_2} : Alg(SPEC_1) \to Alg(SPEC_2)$$

that transforms each $SPEC_1$-datatype A into a $SPEC_2$-datatype that is freely generated by $SPEC_2$ w.r.t. A. The free construction is said to be *strongly persistent at* A if it leaves A unchanged, i.e.,

$$[FREE_{SPEC_1 \to SPEC_2}(A)]_{SPEC_1} = A$$

and *strongly persistent* if it is strongly persistent at all $A \in Alg(SPEC_1)$.

We now define an amalgamated sum operation that combines given datatypes $A_1 \in Alg(SPEC_1)$ and $A_2 \in Alg(SPEC_2)$. The complicating factor here is that $SPEC_1$ and $SPEC_2$ may have a nonempty intersection $SPEC_{12}$, in which case we require the $SPEC_{12}$ reduct of A_1 to be equal to the $SPEC_{12}$ reduct of A_2. A proof of the following proposition concerning the amalgamated sum appears in [EM85]; we present a version of it here simply to facilitate further constructions. Note that, for the purposes of this proposition, the only relevant aspect of $SPEC_{12}$ is its signature.

Proposition 1 (Amalgamated Sum) *Let* $SPEC_1$ *and* $SPEC_2$ *be algebraic specifications,*

$$SPEC_{12} = SPEC_1 \cap SPEC_2$$

and

$$SPEC = SPEC_1 \cup SPEC_2$$

1. *Given datatypes* $A_1 \in Alg(SPEC_1)$ *and* $A_2 \in Alg(SPEC_2)$ *with common subdatatype*

   $$A_{12} = [A_1]_{SPEC_{12}} = [A_2]_{SPEC_{12}} \in Alg(SPEC_{12})$$

   there is a datatype $A_1 + A_2 \in Alg(SPEC)$, *called the amalgamated sum of* A_1 *and* A_2, *such that* $[A_1 + A_2]_{SPEC_1} = A_1$ *and* $[A_1 + A_2]_{SPEC_2} = A_2$.

2. *Conversely, for each datatype* $A \in Alg(SPEC)$ *there are unique datatypes* $A_1 \in Alg(SPEC_1)$ *and* $A_2 \in Alg(SPEC_2)$ *such that* $A = A_1 + A_2$.

Proof:

1. Let $SPEC = (S, OP, E)$, then $A = A_1 + A_2$ is defined as follows. For each $s \in S$, $A_s = A_{i_s}$ for some arbitrary $SPEC_i$ that includes s. A_s is well-defined here because there must be at least one such $SPEC_i$ and if both specifications include s, then $SPEC_{12}$ includes s and

   $$A_{1_s} = [A_1]_{SPEC_{12_s}} = [A_2]_{SPEC_{12_s}} = A_{2_s}.$$

   Similarly, for each $N \in OP$, $N_A = N_{A_i}$ for some arbitrary $SPEC_i$ that includes N. Finally, A satisfies every equation $e \in E$ since there must be at least one $SPEC_i$ that includes e and A_i satisfies e.

2. We first argue that if $A_1 = [A]_{SPEC_1}$ and $A_2 = [A]_{SPEC_2}$ then $A = A_1 + A_2$. It suffices to show that

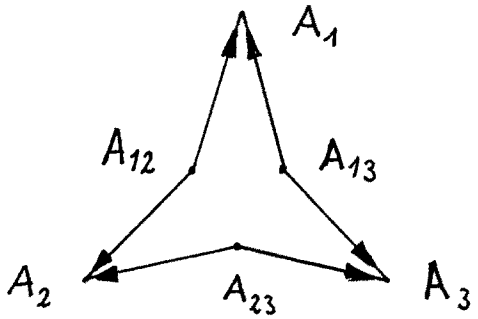
$$[A_1]_{SPEC_{12}} = [[A]_{SPEC_1}]_{SPEC_{12}} = [A]_{SPEC_{12}} = [[A]_{SPEC_2}]_{SPEC_{12}} = [A_2]_{SPEC_{12}}$$

To show uniqueness, suppose $A = B_1 + B_2$, then $A_i = [A]_{SPEC_i} = B_i$.

□

We sometimes write $A_1 +_{A_{12}} A_2$ to make explicit the common subdatatype over which the sum is constructed.

The above construction can be generalized in a straightforward way to allow the "star" amalgamation $\sum_{i=1}^{n} A_i$ of a collection A_i of datatypes for $1 \leq i \leq n$, each associated with an algebraic specification $SPEC_i$ with $SPEC_{ij} = SPEC_i \cap SPEC_j$ and common subdatatypes $A_{ij} \in SPEC_{ij}$ for $1 \leq i < j \leq n$. The "star diagram" of embeddings for the case $n = 3$ has the following form.



4 Abstract Program Modules

In this section, we define abstract program modules. Sections 4.1 and 4.2 introduce the syntax and semantics, respectively, of abstract program modules. Section 4.3 gives a final comparison between abstract and concrete program modules.

4.1 Syntax of Abstract Program Modules

In section 2 we used the following syntax for abstract program modules.

```
MODULE <name>
   EXPORT
      IMPORT <name list>
      <definition of algebraic specifications>
   BODY
      IMPORT <name list>
      <definition of algebraic specifications>
END
```

The following definition introduces an abstract form of this syntax.

Definition 1 (Abstract Program Modules) *An abstract program module is a tuple*

$$(N, EXPIMP_N, EXPDEF_N, BODIMP_N, BODDEF_N)$$

where N is a module name, $EXPIMP_N$ and $BODIMP_N$ are sets of module names, and $EXPDEF_N$ and $BODDEF_N$ are (possibly incomplete) algebraic specifications.

$$COMIMP_N = EXPIMP_N \cup BODIMP_N$$

is called the combined import *of N.*

We often refer to "the abstract program module with name N" more simply as "the module N".

Definition 2 (Modular Systems) *A modular system over a set $NAMES$ of module names is a family*

$$(N, EXPIMP_N, EXPDEF_N, BODIMP_N, BODDEF_N)_{N \in NAMES}$$

of abstract program modules.

Modular systems are expected to follow certain naming conventions for resources, i.e., sorts and operations, as outlined in section 2.

Definition 3 (Naming Conventions) *A modular system satisfies the naming conventions if*

1. *all resources declared within the same abstract program module have pairwise different names,*
2. *every reference to a resource within an abstract program module in which that resource is not declared is prefixed by the name of an abstract program module in which that resource is declared, and*
3. *a name is used as a prefix only in abstract program modules in which that name is imported.*

A modular system is also expected to follow certain "reference" conditions to ensure that its semantics is well-defined. The first condition states that the system must be hierarchically structured.

Definition 4 (Reference Condition I) *A modular system satisfies reference condition I if there is no recursive usage (w.r.t. importation) among abstract program modules.*

Definition 5 (Expanded Components) *For every module N in a modular system that satisfies reference condition I,*

- *the* expanded export specification *of N, denoted EXP_N, is the union of $EXPDEF_N$ and EXP_M for all $M \in EXPIMP_N$,*
- *the* expanded body specification *of N, denoted BOD_N, is the union of EXP_N, $BODDEF_N$, and EXP_M for all $M \in BODIMP_N$, and*
- *the* expanded import specification *of N, denoted IMP_N, is the union of EXP_M for all $M \in COMIMP_N$.*

The base cases in the above recursive definition are modules with empty combined import; reference condition I ensures that this definition is meaningful. The second reference condition states that the expanded components of a modular system must be complete algebraic specifications.

Definition 6 (Reference Condition II) *A modular system over $NAMES$ that satisfies reference condition I satisfies reference condition II if for every module $N \in NAMES$, EXP_N and BOD_N are complete algebraic specifications.*

Note that this condition is satisfied if every resource referenced in $EXPDEF_N$ is declared in EXP_N and every resource referenced in $BODDEF_N$ is declared in BOD_N. This trivially implies that IMP_N is a complete algebraic specification.

4.2 Semantics of Abstract Program Modules

We now inductively build up the semantics of the modules in a modular system. We assume this system satisfies the naming conventions and the reference conditions. As a first step, we define the meaning of *base modules*, i.e., modules N with empty combined import $COMIMP_N$. In this case, $EXP_N = EXPDEF_N$ and $BOD_N = EXPDEF_N \cup BODDEF_N$. We take the meaning A_N of the body in isolation to be the initial datatype w.r.t. BOD_N. The semantics of N is taken to be the restriction of this datatype according to the export specification.

Definition 7 (Semantics of Base Modules) *Let N be a module with empty combined import $COMIMP_N$. The semantics $[\![N]\!]$ of N is defined by*

$$\begin{array}{rcl} A_N & = & T_{BOD_N} \\ [\![N]\!] & = & [A_N]_{EXP_N} \end{array}$$

We now give the semantics of *compound modules*, i.e., modules N with non-empty combined import $COMIMP_N$. In analogy to the base case, we want the meaning of N to be the restriction of the meaning A_N of the body according to the export specification. We define A_N in terms of the meanings of the imported modules, in particular, as the free construction $FREE_{IMP_N \to BOD_N}$ applied to those meanings.

Definition 8 (Semantics of Compound Modules) *Let N be a module with non-empty combined import $COMIMP_N$. The semantics $[\![N]\!]$ of N is defined by*

$$\begin{array}{rcl} A_N & = & FREE_{IMP_N \to BOD_N}(\sum_{M \in COMIMP_N} [\![M]\!]) \\ [\![N]\!] & = & [A_N]_{EXP_N} \end{array}$$

Here, $\sum_{M \in COMIMP_N} [\![M]\!]$ is the star amalgamation of datatypes as described in section 3.

Certain additional requirements are necessary to ensure that the semantics of a module is well-defined and well-behaved. The first problem here is that the star amalgamation $\sum_{M \in COMIMP_N} [\![M]\!]$ is defined only if

$$[[\![M1]\!]]_{S12} = [[\![M2]\!]]_{S12}$$

for all pairs $M1, M2 \in COMIMP_N$, where $S12 = EXP_{M1} \cap EXP_{M2}$. The second problem is that inappropriate declarations and equations in a module can change the

semantics of its imported modules. For example, the equation `0=1` will collapse the elements of the sort `Cardinal` to a single value. Similarly, the declaration `foo : -> Cardinal` with no accompanying equations will add a new element to `Cardinal`. Formally speaking, we want the semantics of a module N to be *import protecting* in the sense that

$$[A_N]_{EXP_M} = [\![M]\!]$$

for all $M \in COMIMP_N$. Note that if N is import protecting, then it is trivially *external* import protecting, i.e., $[[\![N]\!]]_{EXP_M} = [\![M]\!]$ for all $M \in EXPIMP_N$. Theorem 1 below shows that the semantics of a module is both defined and import protecting if the free construction is strongly persistent whenever it is applied.

Definition 9 (Internal Correctness) *A modular system over $NAMES$ is internally correct if*

1. *it satisfies the naming conventions,*
2. *it satisfies the reference conditions, and*
3. *the free construction for every abstract program module $N \in NAMES$ is strongly persistent whenever it is applied, i.e., $[FREE_{IMP_N \to BOD_N}(A)]_{IMP_N} = A$.*

Theorem 1 (On Definedness and Import Protection) *The semantics of every abstract program module in an internally correct modular system is defined and import protecting.*

Proof: We sketch the proof by induction. The hypothesis is trivially true for base modules N, since $COMIMP_N$ is empty. Now let N be a compound module. By the naming conventions, for each pair $M1, M2 \in COMIMP_N$, $S12 = EXP_{M1} \cap EXP_{M2}$ consists of the union of the expanded export specifications of some re-exported modules. By the inductive hypothesis, $M1$ and $M2$ are import protecting and therefore $[[\![M1]\!]]_{S12} = [[\![M2]\!]]_{S12}$. Thus, the condition which ensures the definedness of the star amalgamation $SA = \sum_{M \in COMIMP_N} [\![M]\!]$ is satisfied and $[\![N]\!]$ is *defined*. From the strong persistency requirement it follows that

$$[FREE_{IMP_N \to BOD_N}(SA)]_{IMP_N} = SA.$$

Since for all $M \in COMIMP_N$, $EXP_M \subseteq IMP_N$ and by theorem 1 on the star-amalgamation $[SA]_{EXP_M} = [\![M]\!]$ we have

$$[FREE_{IMP_N \to BOD_N}(SA)]_{EXP_M} = [\![M]\!].$$

Therefore the semantics of N is *import protecting.* □

4.3 Comparison with Concrete Program Modules

It is instructive to draw an analogy between the compilation of concrete program modules and the assignment of semantics to abstract program modules. The creation of executable code for a compound concrete program module entails two steps; compilation of the module using compiled import definition modules and linking of the code using compiled import implementation modules. The result of the first step is modeled by the function

$$f_N(A) = [FREE_{IMP_N \to BOD_N}(A)]_{EXP_N}$$

over $A \in Alg(IMP_N)$, which uses only the export parts of the imported modules. The result of the second step is modeled by the datatype

$$[\![N]\!] = f_N(\sum_{M \in COMIMP_N} [\![M]\!])$$

which uses the bodies of the imported modules. External import protection, i.e.,

$$[[\![N]\!]]_{EXP_M} = [\![M]\!]$$

states that the behavior of module $M \in EXPIMP_N$ should not change when it is linked into N. This issue does not arise in conventional programming languages because the standard constructs ensure strong persistency.

5 Algebraic Module Specifications

In this section, we introduce algebraic module specifications as defined in [BEPP87, EM90] and compare them with abstract program modules. Within the context of our discussions, there are three primary differences between these two concepts. First, the import part of a module specification simply describes its required resources rather than naming specific modules to provide those resources. Second, compound module specifications are built up using a variety of operators, including composition $\bullet$, i.e., import/export matching, and union $\oplus$. Third, a module specification has a single import part that services its body and, indirectly, also its export part. Informally speaking, an abstract program module

$$(N, \{N1\}, EXPDEF_N, \{N2\}, BODDEF_N)$$

can be translated into a module specification $trans(N)$ as follows. Let module specification MOD have import part, export part and body equal to the expanded import, export, and body specifications, respectively, of N, then

$$trans(N) = MOD \bullet (trans(N1) \oplus trans(N2)).$$

Our presentation of module specifications is a special case of the more general theory developed in [EM90]. In the general theory, the parts of a module specification may be connected by arbitrary specification morphisms; in this paper, specification inclusions are always assumed. More significantly, the general theory supports a fourth module component, called the *parameter part*, that models generic parameterization. Many of the results presented in this section are simple specializations of more general results presented in [EM90].

5.1 Syntax and Semantics of Module Specifications

Module specifications have the following abstract syntax.

Definition 10 (Module Specifications) *A module specification MOD is a tuple*

$$(IMP, EXP, BOD)$$

of complete algebraic specifications where $IMP \subseteq BOD$ and $EXP \subseteq BOD$.

IMP, EXP, and BOD are called the import part, export part, and body, respectively, of MOD.

The meaning $[\![MOD]\!]$ of module specification MOD is taken to be a function from import datatypes to export datatypes.

Definition 11 (Semantics of Module Specifications) *Let*

$$MOD = (IMP, EXP, BOD)$$

be a module specification, then $[\![MOD]\!] : Alg(IMP) \to Alg(EXP)$ *is given by*

$$[\![MOD]\!](A) = [FREE_{IMP \to BOD}(A)]_{EXP}$$

for all $A \in Alg(IMP)$.

Note that if the import part IMP is empty, then for the empty datatype $\emptyset$,

$$FREE_{IMP \to BOD}(\emptyset) = T_{BOD},$$

the initial datatype w.r.t. BOD.

Again, we must worry about whether the meanings of import datatypes are changed. We say that MOD is *pointwise correct* for $A \in Alg(IMP)$ if the free construction is strongly persistent at A, i.e.,

$$[FREE_{IMP \to BOD}(A)]_{IMP} = A$$

and that MOD is *correct* if it is pointwise correct for all $A \in Alg(IMP)$. Note that MOD is always correct if IMP is empty. Generally speaking, useful and correct module specifications cannot be formulated within our restricted syntax. The problem is that correctness may hold only for import datatypes of some appropriate form, e.g., where the sort `Boolean` has distinct elements `true` and `false`. One solution to this problem is to place high-level constraints on the form of import and export datatypes; this approach is investigated in [EW86, EFPPB86, EM90]. We adopt the more "programming-language-oriented" solution of requiring correctness only for the specific import datatypes used in a given construction.

5.2 Operations on Module Specifications

The composition operation $\bullet$ connects the export part of one module specification to the import part of another.

Definition 12 (Composition) *Let* MOD_1 *and* MOD_2 *be module specifications such that* $IMP_1 \subseteq EXP_2$ *and* $BOD_1 \cap BOD_2 = IMP_1$.

$$MOD_1 \bullet MOD_2 = (IMP_2, EXP_1, BOD_1 \cup BOD_2)$$

The following theorem shows that composition preserves correctness and that the semantics of a composed module specification can be expressed in terms of the semantics of its parts.

Theorem 2 (Correctness and Compositionality of Composition) *Let the module specifications* MOD_1 *and* MOD_2 *be correct and let* $MOD_1 \bullet MOD_2$ *be defined.*

1. $MOD_1 \bullet MOD_2$ *is correct.*
2. *For all* $A \in Alg(IMP_2)$, $[\![MOD_1 \bullet MOD_2]\!](A) = [\![MOD_1]\!]([[\![MOD_2]\!](A)]_{IMP_1})$

Proof: Specialization of corresponding theorems in [EM90]. □

A pointwise formulation of the above theorem follows as a corollary.

We now define the union operation $\oplus$. As a first step, we introduce the notion of a submodule.

Definition 13 (Submodules) MOD_1 *is a submodule of* MOD_2 *if*

1. *each component of* MOD_1 *is a subspecification of the corresponding component of* MOD_2, *and*
2. *the free constructions of* MOD_1 *and* MOD_2 *are compatible, i.e., for all* $A \in Alg(IMP_2)$, $FREE_{IMP_1 \to BOD_1}([A]_{IMP_1}) = [FREE_{IMP_2 \to BOD_2}(A)]_{BOD_1}$

In the following definition, the syntactic union $\cup$ (resp. intersection $\cap$) of two module specifications is taken to be the union (resp. intersection) of each of their components.

Definition 14 (Union) *Let* MOD_1 *and* MOD_2 *be module specifications such that their intersection* $MOD_1 \cap MOD_2$ *is a submodule of both* MOD_1 *and* MOD_2.

$$MOD_1 \oplus MOD_2 = MOD_1 \cup MOD_2$$

The following theorem shows that union preserves correctness and that the semantics of the union of two module specifications can be expressed in terms of the semantics of its parts.

Theorem 3 (Correctness and Compositionality of Union) *Let the module specifications* MOD_1 *and* MOD_2 *be correct and let* $MOD_1 \oplus MOD_2$ *be defined. Furthermore, let* $MOD_{12} = MOD_1 \cap MOD_2$.

1. $MOD_1 \oplus MOD_2$ *is correct.*
2. *For all* $A \in Alg(IMP_1 \cup IMP_2)$,

   $$[\![MOD_1 \oplus MOD_2]\!](A) = [\![MOD_1]\!]([A]_{IMP_1}) +_{[\![MOD_{12}]\!]([A]_{IMP_{12}})} [\![MOD_2]\!]([A]_{IMP_2})$$

   where $+$ *denotes the amalgamated sum of datatypes.*

Proof: Specialization of corresponding theorems in [EM90]. □

A pointwise formulation of the above theorem follows as a corollary.

5.3 Comparison with Abstract Program Modules

In this section, we show how a system of abstract program modules can be translated into a system of module specifications. For simplicity, we focus on *small* modules, where each import part of a small module contains exactly one module name, as in the introduction to this section. This case possesses enough structure to demonstrate how the translation works in principle. The generalization to arbitrary abstract program modules is straightforward but is technically more complex.

Definition 15 (Translation Function *trans*) *Given an internally correct modular system over $NAMES$ containing only base modules and small modules, each $N \in NAMES$ is mapped to a module specification $trans(N)$ as follows. For every base module*

$$(N, \emptyset, EXPDEF_N, \emptyset, BODDEF_N),$$

let

$$trans(N) = (\emptyset, EXP_N, BOD_N)$$

and for every small module

$$(N, \{N1\}, EXPDEF_N, \{N2\}, BODDEF_N),$$

let

$$trans(N) = MOD \bullet (trans(N1) \oplus trans(N2))$$

where $MOD = (IMP_N, EXP_N, BOD_N)$.

Note that the import part of the module specification $trans(N)$ is empty for all N: this is clearly true for base modules and the import part of $trans(N) = MOD \bullet (trans(N1) \oplus trans(N2))$ is equal to the union of the import parts of $trans(N1)$ and $trans(N2)$.

To improve the readability of the following theorem, we add subscripts to our semantic functions: $[\![N]\!]_P$ is the meaning of abstract program module N and $[\![MOD]\!]_S$ is the meaning of module specification MOD. The following theorem shows that the semantics $[\![N]\!]_P$ of abstract program module N is equal to the semantics $[\![trans(N)]\!]_S$ of the module specification $trans(N)$.

Theorem 4 (Correctness of *trans*) *Given a modular system as in definition 15 above, then*

$$[\![trans(N)]\!]_S = [\![N]\!]_P$$

for all $N \in NAMES$.

Proof: As a first step, we sketch the proof of several definedness and correctness lemmas concerning

$$trans(N) = MOD \bullet (trans(N1) \oplus trans(N2))$$

where $MOD = (IMP_N, EXP_N, BOD_N)$.

1. MOD is well-defined (see definition 10). This fact follows from the definition of IMP_N, EXP_N, and BOD_N, and the reference conditions.
2. $trans(N1) \oplus trans(N2)$ is well-defined (see definition 14). This fact follows from the fact that $N1$ and $N2$ are import protecting, thus, the free constructions of $trans(N1)$ and $trans(N2)$ are compatible.
3. $MOD \bullet (trans(N1) \oplus trans(N2))$ is well-defined (see definition 12). This fact follows directly from the definitions of $\oplus$, IMP_N, and BOD_N.
4. Every occurring module specification is pointwise correct at the point it is applied. $trans(N1)$ and $trans(N2)$ are trivially pointwise correct. Therefore, $trans(N1) \oplus trans(N2)$ is pointwise correct by theorem 3. The fact that MOD is pointwise correct for $[\![trans(N1) \oplus trans(N2)]\!]_S$ follows from the construction of MOD and the fact that N is import protecting.

We now show that the theorem holds by induction over the structure of N. For the base case, consider base modules N.

$$\begin{aligned}
[\![trans(N)]\!]_S &= [\![(\emptyset, EXP_N, BOD_N)]\!]_S \\
&= [\![(\emptyset, EXPDEF_N, EXPDEF_N \oplus BODDEF_N)]\!]_S \\
&= [FREE_{\emptyset \to EXPDEF_N \oplus BODDEF_N}(\emptyset)]_{EXPDEF_N} \\
&= [T_{EXPDEF_N \oplus BODDEF_N}]_{EXPDEF_N} \\
&= [\![N]\!]_P
\end{aligned}$$

For the inductive step, consider compound modules N.

$$\begin{aligned}
[\![trans(N)]\!]_S &= [\![MOD \bullet (trans(N1) \oplus trans(N2))]\!]_S \\
&= [\![MOD]\!]_S([\![trans(N1) \oplus trans(N2)]\!]_S) && \text{4 above, theorem 2} \\
&= [\![MOD]\!]_S([\![trans(N1)]\!]_S + [\![trans(N2)]\!]_S) && \text{4 above, theorem 3} \\
&= [\![MOD]\!]_S([\![N1]\!]_P + [\![N2]\!]_P) && \text{inductive hypothesis} \\
&= [FREE_{IMP_N \to BOD_N}([\![N1]\!]_P + [\![N2]\!]_P)]_{EXP_N} \\
&= [\![N]\!]_P
\end{aligned}$$

□

6 Concluding Remarks

This paper has studied the relationship between module specifications and programming language modules. Our approach was to introduce an intermediate algebraic concept of abstract program modules, which structurally model concrete program modules, and then to relate systems of abstract program modules to systems of module specifications. The primary result of this paper, theorem 4, shows that every system of abstract program modules can be translated into a system of module specifications with an equivalent structure and semantics. A practical consequence of this fact is that one need never worry that certain (desirable) implementations are unreachable from the specification level. On the other hand, module specifications are clearly more expressive than abstract program modules. This suggests that the transformational development of a system from module specifications to program modules must entail a phase where the module specifications are suitably restructured and refined.

Module specifications and concrete program modules both support the notion of generic parameterization. We plan to extend abstract program modules to include this notion and study its impact on the ability of module specifications to model abstract program modules.

References

[BEPP87] E.K. Blum, H. Ehrig, and F. Parisi-Presicce. Algebraic specification of modules and their basic interconnection. *JCSS*, 34, 1987.

[EFPPB86] H. Ehrig, W. Fey, F. Parisi-Presicce, and E.K. Blum. Algebraic theory of module specifications with constraints. In *Proceedings MFCS*. Springer-Verlag, LNCS 233, 1986.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I.* Springer Verlag, Berlin, 1985.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification II.* Springer Verlag, Berlin, 1990.

[EW86] H. Ehrig and H. Weber. *Programming in the large with algebraic module specifications.*, pages 675–684. Information Processing 86. North-Holland, Amsterdam, 1986.

[FGJM85] K. Futatsugi, J.A. Goguen, J.P. Jouannaud, and J. Meseguer. Principles of obj2. In *Proceedings 12th ACM Symposium on Principles of Programming Languages*, 1985.

[Gau84] M.-C. Gaudel. A first introduction to pluss. In *Proceedings ALYEY Workshop on Formal Specification*, 1984. Swendon.

[GTW76] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. Technical Report RC 6487, IBM, 1976.

[HMM86] R. Harper, D.B. MacQueen, and R. Milner. Standard ml. Technical Report ECS-LFCS-86-2, University of Edinburgh, 1986.

[LZ74] B. Liskov and S. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9, 1974.

[Par72a] D.C. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058, 1972.

[Par72b] D.C. Parnas. A technique for software module specification with examples. *CACM*, 15(5):330–336, 1972.

[PDN82] R. Prieto-Diaz and J. Neighbors. Module interconnection languages: a survey. Technical Report ICS Technical Report 189, C.S. Dept. UC Irvine, 1982.

[Spr81] Springer Verlag LNCS 106, Berlin. *The Programming Language Ada, Reference Manual*, 1981.

[ST86] D. Sannella and A. Tarlecki. Extended ml: an institution independent framework for formal program development. In *Proceedings Workshop on Category Theory and Computer Programming*, pages 364–389. LNCS 240, 1986.

[TWW78] J.W. Thatcher, E.G. Wagner, and J.B. Wright. Data type specification: parameterization, and the power of specification techniques. In *10th Symp. on Theory of Computing*, 1978.

[Wir82] N. Wirth. *Programming in Modula-2.* Springer Verlag, Berlin, 1982.